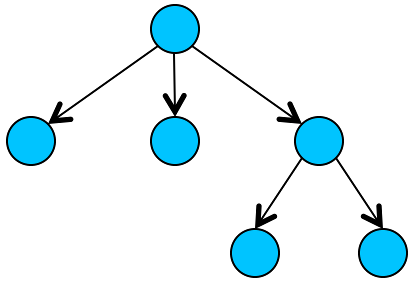
8. Trees

- Motivation
- Tree Terminology
- Binary Search Trees
- BST Declaration
- BST Print
- BST Search
- BST Insert
- BST Delete
- BST Balance
- BST Analysis
- Tree Discussion

Motivation

- A tree is an ADT that stores data in a hierarchical way, much like a family tree



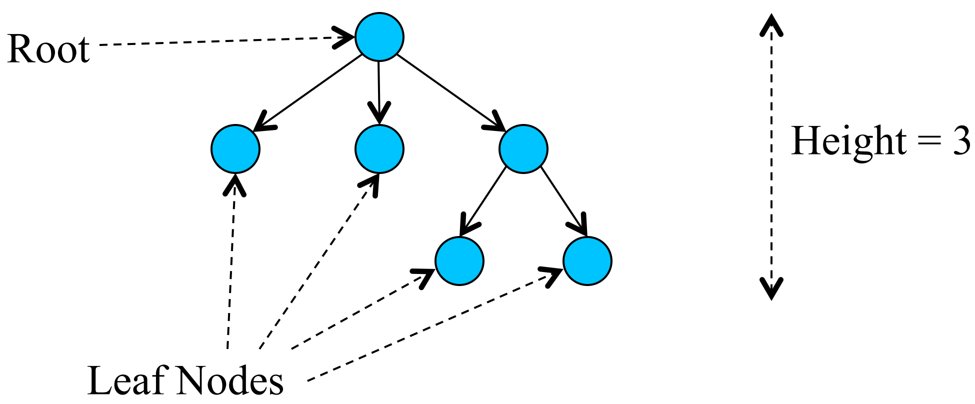
- For node has most one <u>parent</u>
- Each node has zero or more <u>children</u>
- Each node has zero or more <u>siblings</u>

Motivation

- Different <u>types</u> of data can be stored in tree nodes depending on needs of the application
  - numbers, characters, strings
  - objects, other ADTs
- When we limit the number of children to two, we have a <u>binary tree</u>
  - useful for quickly storing and retrieving data
  - also used to represent arithmetic expressions

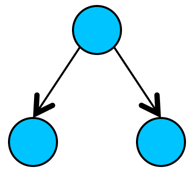
Tree Terminology

- Root – top of tree, has no parent
- Leaf – bottom of tree, has no children
- Height – the number of nodes on the longest path from leaf node to root node

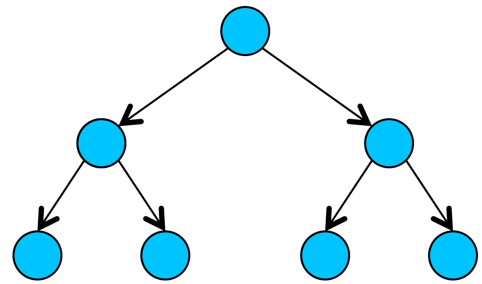


Tree Terminology

- Empty – tree with zero nodes
- Full – binary tree with all leaf nodes at level h and all other nodes have 2 children



Full tree, height 2



Full tree, height 3

Tree Terminology

- How many nodes N can we store in a full binary tree of height h?

  $h = 1, N = 1$

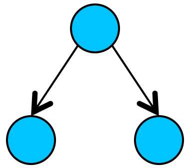
  $h = 2, N = 1+2 = 3$

  $h = 3, N = 1+2+4 = 7$

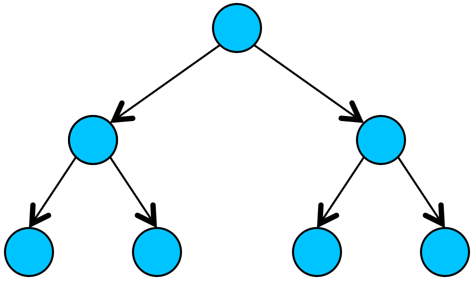
  $h = 4, N = 1+2+4+8 = 15$

  …

  $N = 1+2+\ldots+2^{h-1} = 2^h-1$



$h=2, N= 2^2-1=3$



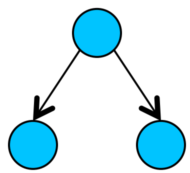
$h=3, N= 2^3-1=7$

Tree Terminology

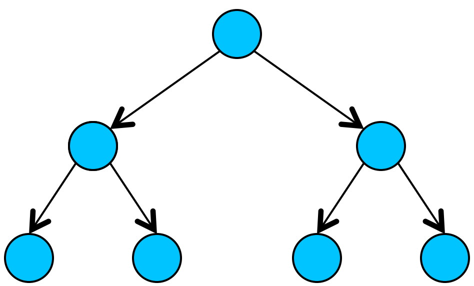
- What is the minimum height of a binary tree that contains N nodes? Assume tree is full.

$$N = 2^h - 1$$

$$h = \log_2(N+1)$$



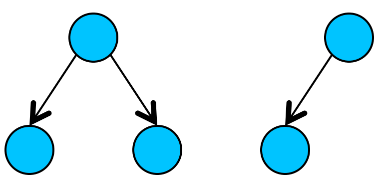
N=3, $h=\log_2 4 = 2$



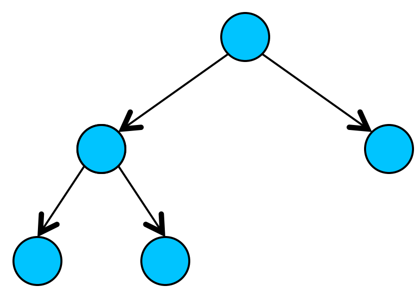
N=7, $h=\log_2 8 = 3$

Tree Terminology

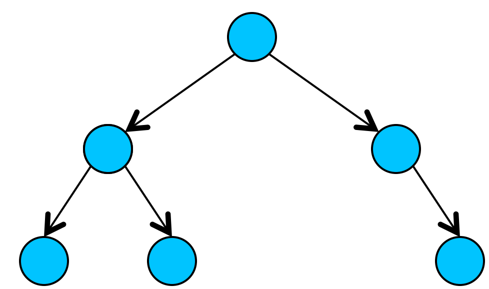
- Complete – a binary tree that is full to level h-1 and all leaf nodes on level h are filled in from <u>left to right</u>



Complete



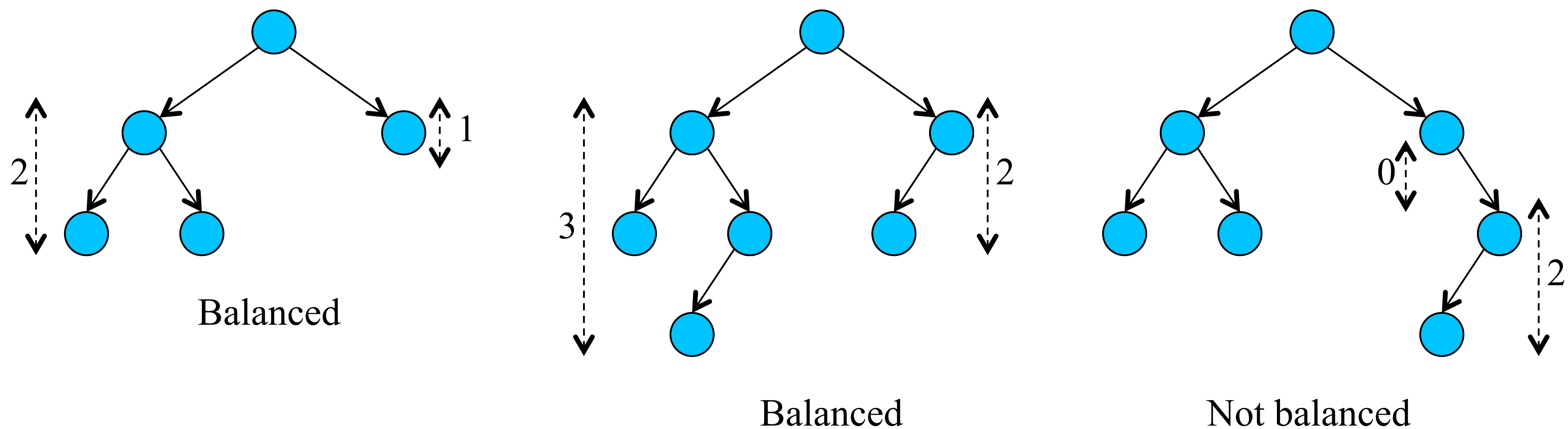
Complete



Not complete

Tree Terminology

- Balanced – a binary tree in which the <u>height</u> of the left and right subtrees of <u>any</u> node in the tree differ by at most one



Binary Search Trees

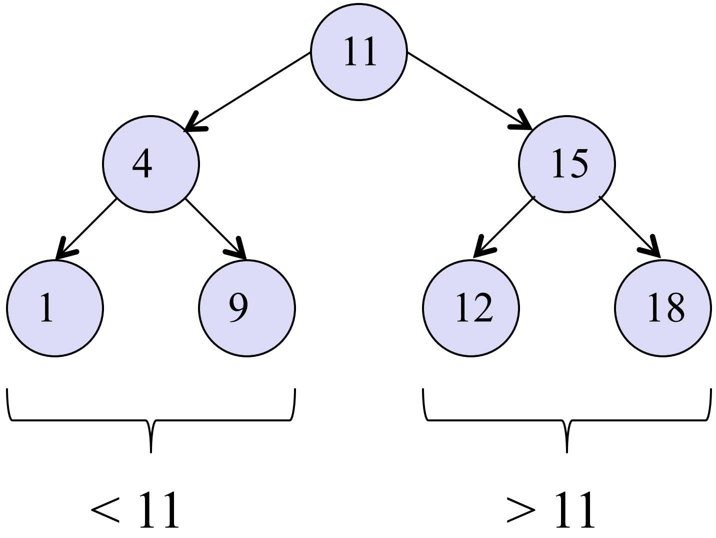
- Consider the task of searching a sorted array of data using binary search

1	4	9	11	12	15	18
0	1	2	3	4	5	6

  - We will always look at at data[3]=11 first
  - If value is <11 we will look at data[1]=4 next
  - If value is >11 we will look at data[5]=15 next
  - This continues until we find the desired value

Binary Search Trees

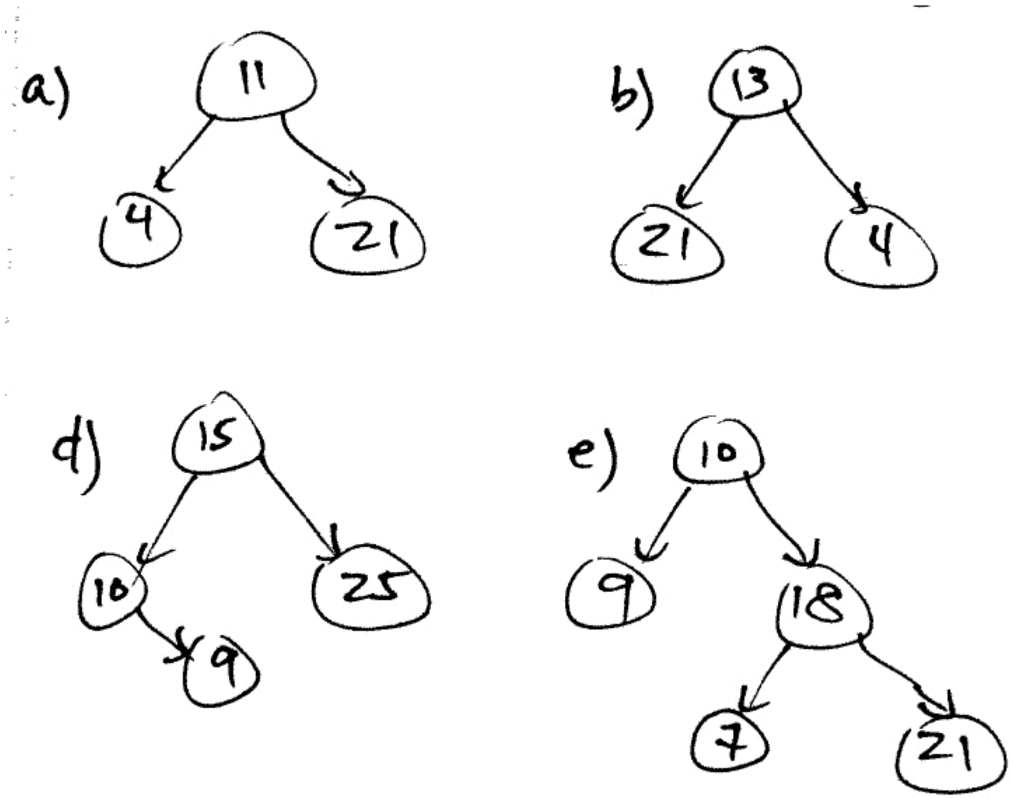
- This sequence of decisions can be stored in a binary search tree (BST)



- All nodes in the left subtree are smaller in value
- All nodes in the right subtree are larger in value
- This is true for all nodes in BST

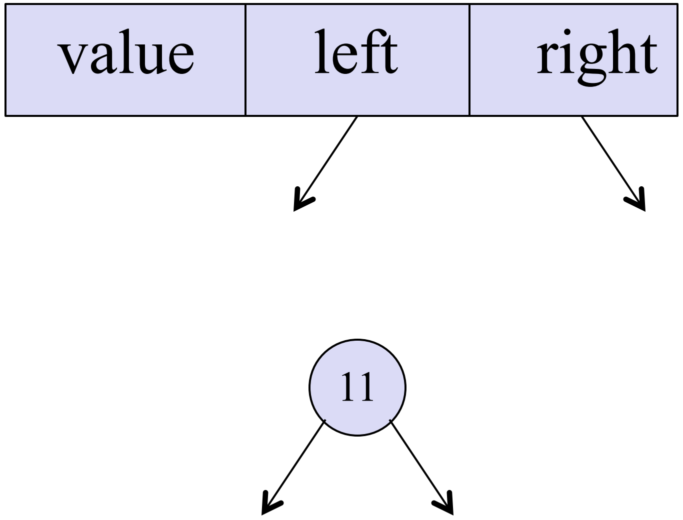
Binary Search Trees

- Which of the following are valid BSTs?



BST Declaration

```
class node
{
public:
   int value;
   node *left;
   node *right;
};
```



BST Declaration

```
class BST
{
public:
   BST();
   ~BST();
   bool search(int value);
   bool insert(int value);
   bool delete(int value);
   void print();
private:
   node *root;
};
```

BST Print

- Assume you are given a valid BST and you want to print all of the values in the tree
- We can do this with a <u>recursive</u> function that visits all of the nodes in the tree
  - We need to pass in a pointer to the root of tree
  - Make recursive calls with left and right pointers
  - The order we visit nodes determines print order

BST Print

```
void print1(node *ptr)
{  // terminating condition
   if (ptr == NULL) return;

   // print left subtree
   print1(ptr->left);

   // print node value
   cout << ptr->value << endl;

   // print right subtree
   print1(ptr->right);
}
```

This will print the data values in sorted order

BST Print

```
void print2(node *ptr)
{  // terminating condition
   if (ptr == NULL) return;

   // print node value
   cout << ptr->value << endl;

   // print left subtree
   print2(ptr->left);

   // print right subtree
   print2(ptr->right);
}
```

This will print the data values in preorder

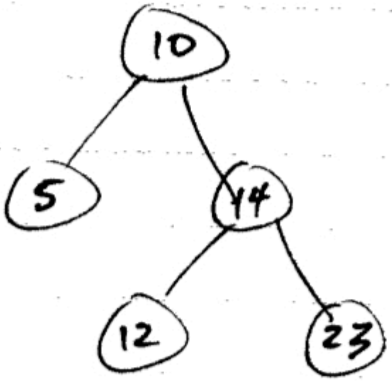
BST Print

```
void print3(node *ptr)
{  // terminating condition
   if (ptr == NULL) return;

   // print left subtree
   print3(ptr->left);

   // print right subtree
   print3(ptr->right);

   // print node value
   cout << ptr->value << endl;
}
```

This will print the data values in postorder

BST Print

- Example with numerical data:



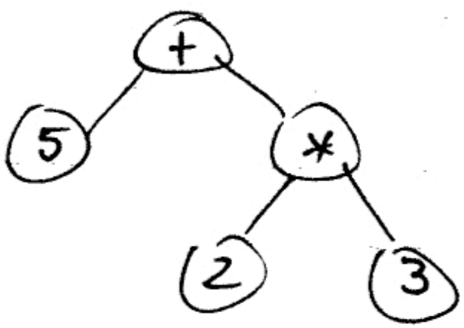
inorder: 5 10 12 14 23

preorder: 10 5 14 12 23

postorder: 5 12 23 14 10

BST Print

- Example with symbolic data:



inorder: 5 + 2 * 3

preorder: + 5 * 2 3

postorder: 5 2 3 * +

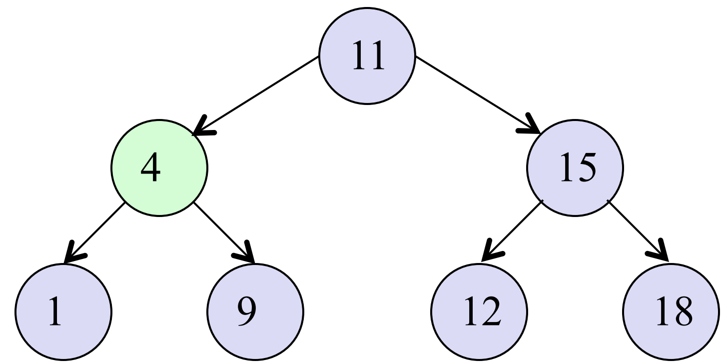
BST Search

- Assume we are given a valid BST and wish to locate a desired value in the tree

- Algorithm:
  - Start ptr at root of tree
  - If node value > desired go to left child
  - If node value < desired go to right child
  - Stop when ptr is null or when value is found

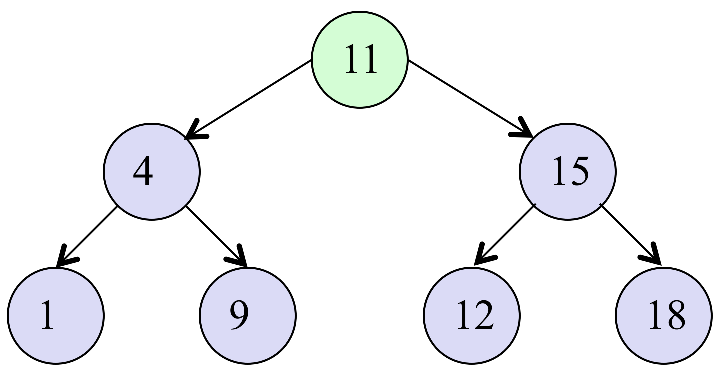
BST Search

- Assume we are searching the BST for the value 9

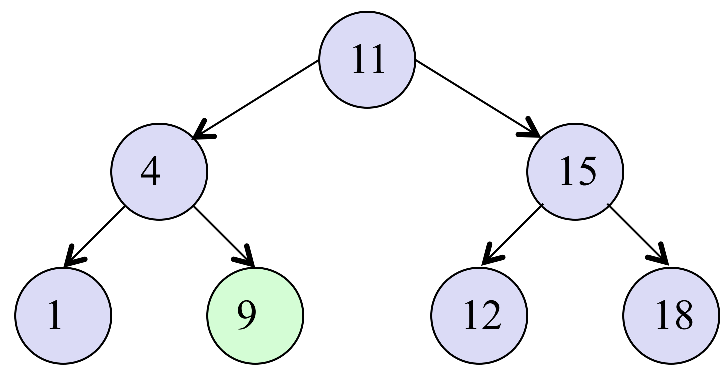


start at root of tree

9 > 4 so go right



9 < 11 so go left

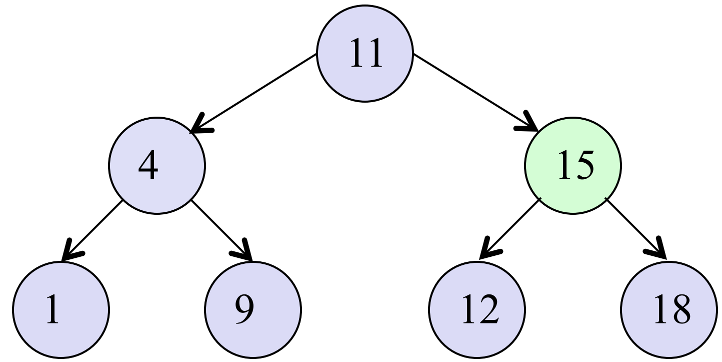


we found the 9 node

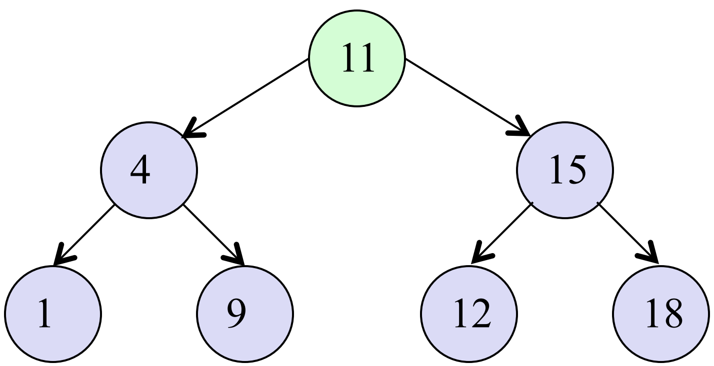
BST Search

- Assume we are searching the BST for the value 13

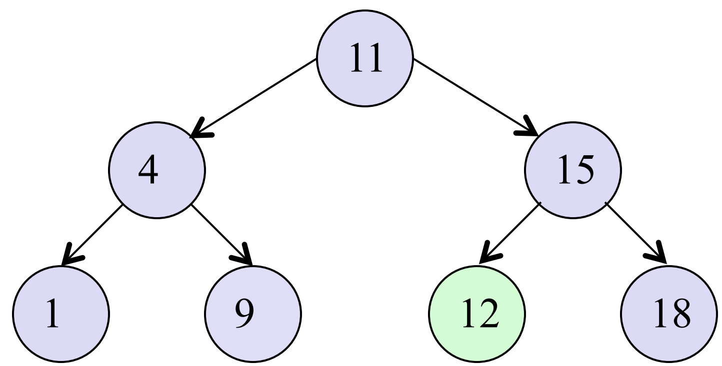
start at root of tree



13 < 15 so go left



13 > 11 so go right



13 > 12 but null pointer to right so the value not found

BST Search

```
bool search_list(int value)
{
   // iteratively search linked list
   node *ptr = head;
   while ((ptr != NULL)&&(ptr->value != value))
   {
      // go to next node
      ptr = ptr->next;
   }
   // return true/false if found or not
   return((ptr != NULL)&&(ptr->value == value));
}
```

BST Search

```
bool search_bst(int value)
{  // iteratively search tree
   node *ptr = root;
   while ((ptr != NULL)&&(ptr->value != value))
   {  // search left or right subtree
      if (ptr->value > value)
         ptr = ptr->left;
      else if (ptr->value < value)
         ptr = ptr->right;
   }
   // return true/false if found or not
   return((ptr != NULL)&&(ptr->value == value));
}
```

BST Search

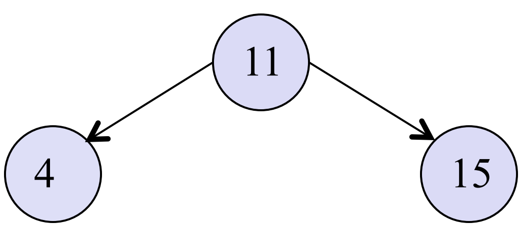
```
bool search_bst(int value, node *ptr)
{  // terminating conditions
   if (ptr == NULL)
      return false;
   else if (ptr->value == value)
      return true;

   // recursively search tree
   if (ptr->value > value)
      return search_bst(value, ptr->left);
   else if (ptr->value < value)
      return search_bst(value, ptr->right);
}
```

BST Search

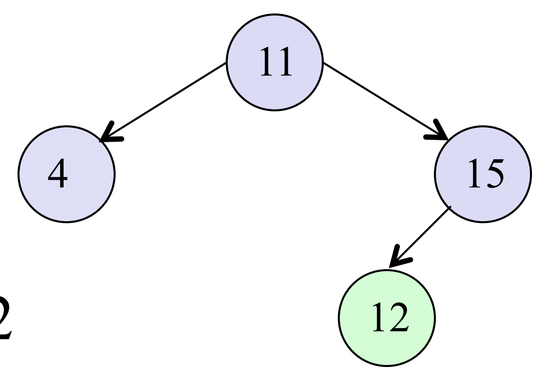
- If we have a balanced BST tree this search algorithm will find data after $O(\log_2 N)$ steps
- If we have a very unbalanced BST tree (like a linked list) search may take $O(N)$ steps
- On average we can expect $O(\log_2 N)$ search

BST Insert

- Assume we are given a BST and must insert a new data value
  - We want to make sure we will still have a valid BST after insertion so we can not insert anywhere
  - Easy method is to search the BST for the desired value and then add a new node at the "dead end"

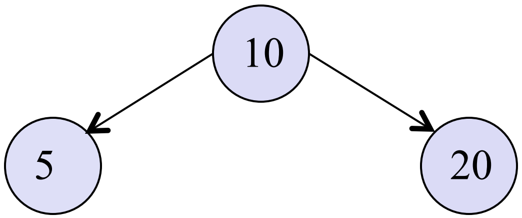


insert value 12

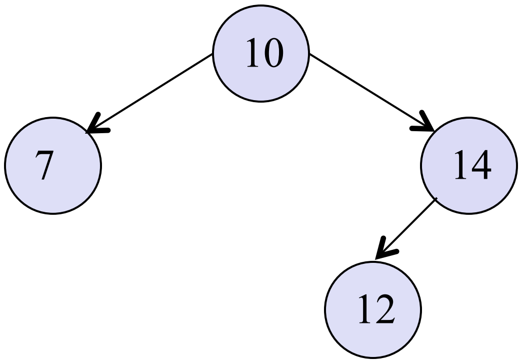


BST Insert

- Examples:



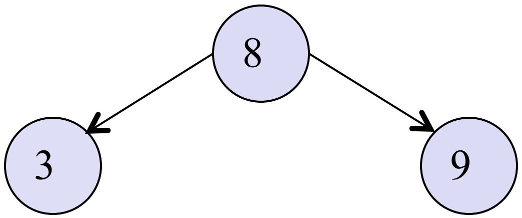
insert 2 and 8



insert 13 and 14

BST Insert

- Examples:



insert 10, 11, and 12

empty tree

insert 42

BST Insert

- With this algorithm we are always inserting a new leaf node (never an internal node)
- The location of new node depends on the values in BST leaves
  - What will happen if we insert N values that are in sorted order?
  - What will happen if we insert N values that are in random order?

BST Insert

```
void insert_bst(int value, node * ptr)
{
   // terminating condition
   if (ptr == NULL)
   {
      // insert node into bst
      ptr = new node;
      ptr->value = value;
      ptr->left = NULL;
      ptr->right = NULL;
   }
…
```

BST Insert

```
…
   // recursive search and insert
   else if (ptr->value > value)
      insert_bst(value, ptr->left);
   else if (ptr->value < value)
      insert_bst(value, ptr->right);
}
```

- What will this function do if we insert duplicate data?
- Do you see any problems with function parameters?

BST Insert

```
void insert_bst(int value, node * & ptr)
{
   // terminating condition
   if (ptr == NULL)
   {
      // insert node into bst
      ptr = new node;
      ptr->value = value;
      ptr->left = NULL;
      ptr->right = NULL;
   }
…
```

BST Insert

```
…
    // recursive search and insert
    else if (ptr->value > value)
        insert_bst(value, ptr->left);
    else if (ptr->value <= value)
        insert_bst(value, ptr->right);
}
```

- This will insert duplicate values into <u>right</u> subtree.

BST Insert

- If we have a balanced BST tree insertion will take $O(\log_2 N)$ steps
- If we have a very unbalanced BST tree insertion may take $O(N)$ steps
- On average we can expect $O(\log_2 N)$ insertion

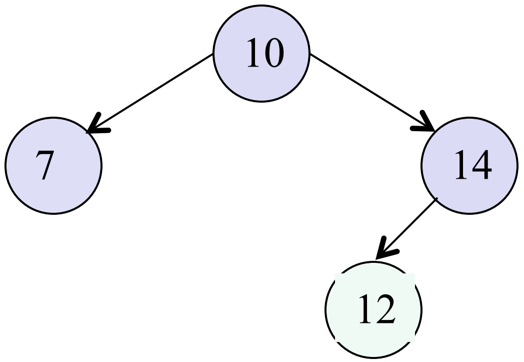
BST Delete

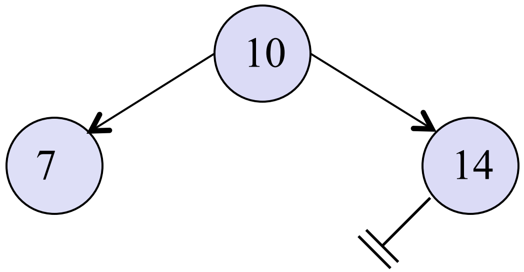
- Assume we are given a valid BST and we wish to delete a node with a given value
- Algorithm:
  - Start at root of tree
  - Search for node to delete from tree
  - Adjust tree pointers to “jump over” deleted node
  - Delete the node

BST Delete

- There are three cases to consider when adjusting tree pointers:

0 children – set pointer to deleted node to null

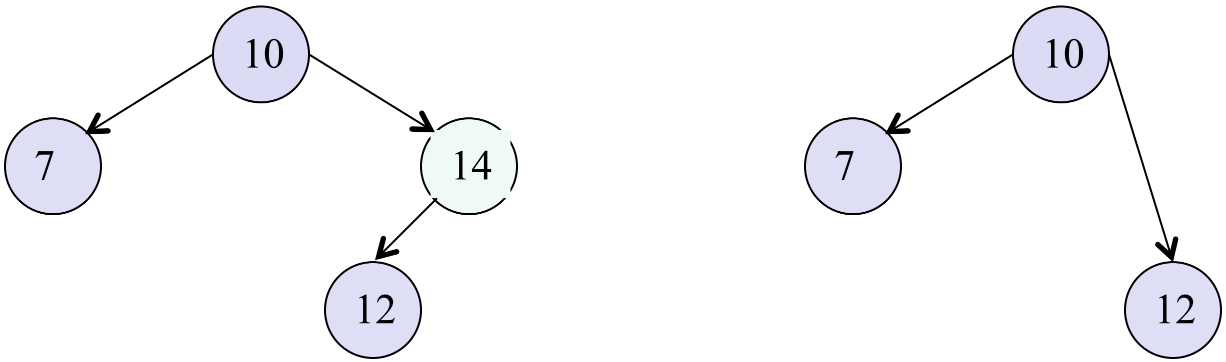




delete value 12

BST Delete

1 child – change pointer in the parent of the deleted node so it points to the child of the deleted node



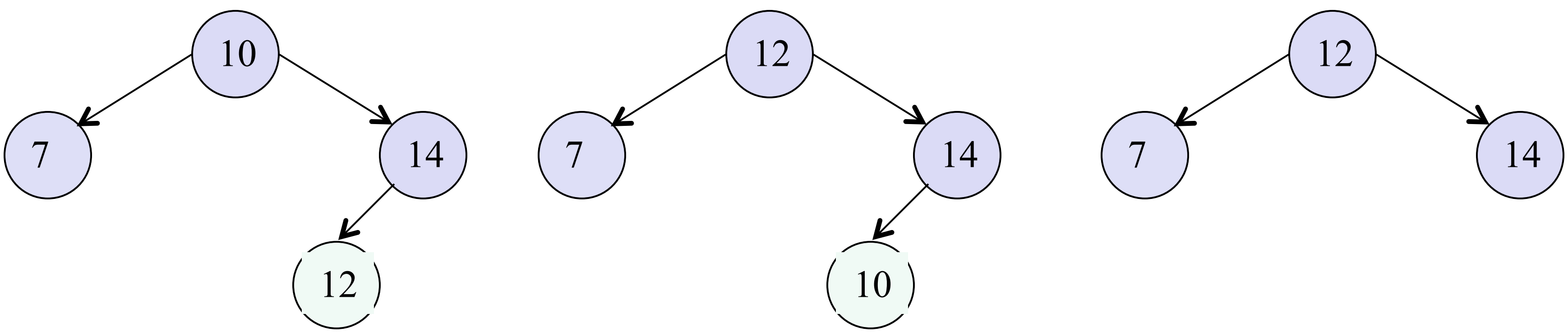
delete value 14

BST Delete

2 children – find <u>left</u> most node in <u>right</u> sub tree

– swap value with node to be deleted

– delete left most node



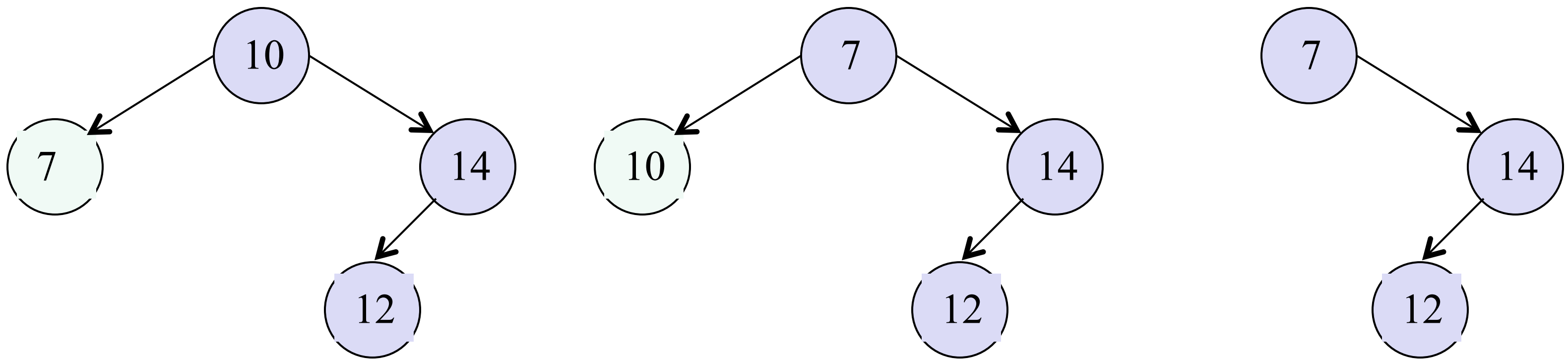
delete value 10

BST Delete

2 children – find <u>right</u> most node in <u>left</u> sub tree

– swap value with node to be deleted

– delete left most node



delete value 10

BST Delete

```
void delete_bst(int value, node * & ptr)
{
   // value not found, so stop
   if (ptr == NULL)
      return;

   // value found, so delete
   else if (ptr->value == value)
      delete_node(ptr);
   …
```

BST Delete

```
    …

    // recursive search left
    else if (ptr->value > value)
        delete_bst(value, ptr->left);

    // recursive search right
    else if (ptr->value < value)
        delete_bst(value, ptr->right);
}
```

BST Delete

```
void delete_node(node * & ptr)
{
   // zero children case
   if ((ptr->left == NULL) && (ptr->right == NULL))
   {
      delete ptr;
      ptr = NULL;
   }
   …
```

BST Delete

```
// one child on right
if ((ptr->left == NULL) && (ptr->right != NULL))
{
   node * temp = ptr;
   ptr = ptr->right;
   delete temp;
}
…
```

BST Delete

```
// one child on left
if ((ptr->left != NULL) && (ptr->right == NULL))
{
   node * temp = ptr;
   ptr = ptr->left;
   delete temp;
}
…
```

BST Delete

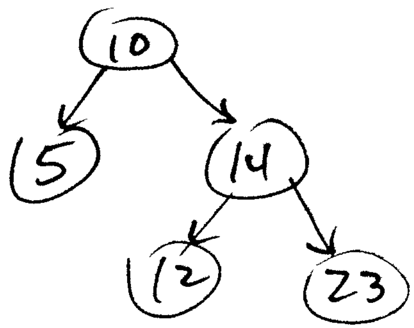
```
// handle two children
if ((ptr->left != NULL) && (ptr->right != NULL))
{
   // find left most node in right sub tree
   node * parent = ptr;
   node * child = parent->right;
   while (child->left != NULL)
   {
      parent = child;
      child = child->left;
   }
…
```

BST Delete

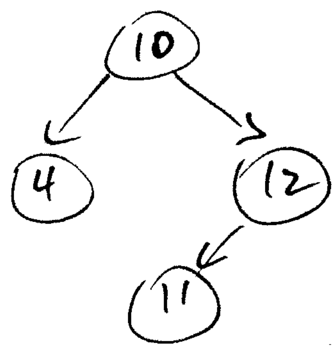
```
…
    // fix pointer to left most node
    if (parent != ptr)
       parent->left = child->right;
    else
       ptr->right = child->right;

    // delete node
    ptr->value = child->value;
    delete child;
  }
}
```

BST Delete

- Examples:



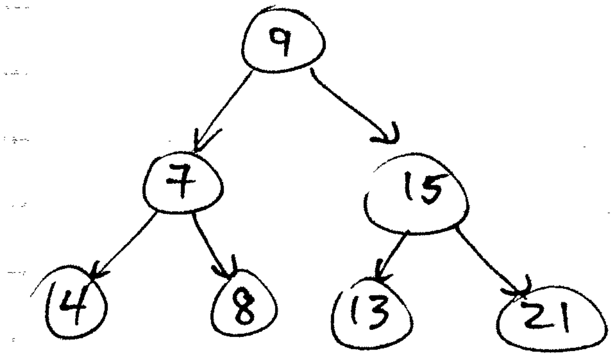
delete 23



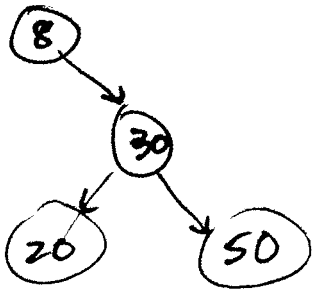
delete 12

BST Delete

- Examples:



delete 9 & 7



delete 8

BST Balance

- TBA

BST Analysis

- TBA

Tree Discussion

- TBA